Block sequencing, partial derivatives and SG7
24th EuroAD Workshop

Dominic Jones

Siemens, London
dominic.jones@cd-adapco.com

2–4 November 2021
Introduction

– Siemen’s Simcenter Star-CCM+ simulation software uses an in-house automatic differentiation tool to differentiate the coupled Naiver-Stokes solver, Spalart Allmaras turbulence model, and a significant amount of related code.

– The in-house route was taken due to the performance requirements of the resultant adjoint code (no taping is used, for example). Secondly, a tool was needed that would not conflict in presence of the PETE vector unrolling engine.

– In the course of applying AD to a large amount of code, a number of techniques have been devised. This talk will look at a couple of them.
Sequencing code blocks (1)
enum class direction : { forward, reverse };

void compute(direction d)
{
    sequence_begin(d)
    {
        std::cout << "block 1" << std::endl;
    }
    sequence_next()
    {
        std::cout << "block 2" << std::endl;
    }
    sequence_next()
    {
        std::cout << "block 3" << std::endl;
    }
    sequence_end();
}
```c
#define sequence_begin(mode) \
{ \
    auto constexpr sequence_offset{__COUNTER__}; \
    auto const sequence_mode{mode}; \
    auto sequence_case{int(0)}; \

    if (mode == direction::reverse) \
        goto sequence_end_label; \

sequence_begin_label: \
    switch (sequence_case) { \
    case 0:
```
#define sequence_next() \
} \
(sequence_mode == direction::reverse) ? \
  sequence_case-- : sequence_case++; \
\ngoto sequence_begin_label; \
\ncase (__COUNTER__ - sequence_offset): \
{
```c
#define sequence_end() \
{ 
    (sequence_mode == direction::reverse) ? 
        sequence_case-- : sequence_case++; 
    goto sequence_begin_label; 
} \
sequence_end_label: \
    default: \
    { 
        auto constexpr size{__COUNTER__ - sequence_offset}; 
        if (sequence_mode == direction::reverse && 
            sequence_case >= 0) 
        { 
            sequence_case = size - 1; 
            goto sequence_begin_label; 
        } 
    } \
break; 
} \
```
Sequencing code blocks (2)
void compute(X1 const &x1, X2 const &x2, Y &y)
{
    X1 t;

    IF (x1 > x2)
    {
        auto t0{x1 - x2};
        t = exp(t0);
    }
    ELSE
    {
        auto t0{x2 - x1};
        t = exp(t0);
    }
    ENDIF

    y = x1 * t;
}
Two lambdas from one

```cpp
template<typename T1, typename T2, typename F>
struct BimodalLambda
{
    F f;
    BimodalLambda(F f) : f(f) { f(T1{}); }
    ~BimodalLambda() { f(T2{}); }
};
```
auto lambda = [&]<typename M>(M mode) // C++20
{
    auto eval = [mode]<typename E>(E const &expr)
    {
        if constexpr (sizeof(mode) == Primal::value)
            return ::eval_primal(expr);
        else
            return ::eval(expr);
    };

    // the 'lambda', e.g.
    {
        auto t0{eval(x1 - x2)}; // must wrap RHS with ‘eval’
        t = exp(t0);
    }
};
Partial differentiation
Avoiding primitives

```cpp
float a = 3, a_drv = 0;
float b = 4, b_drv = 0;
float r_drv = 1;

Drv<Primal, float> a_{a, a_drv}; // passive but not primitive
Drv<Adjoint, float> b_{b, b_drv};
Drv<Adjoint, float&> r_{r_drv};

{
    auto a1 = drv::cos(a_); // non-primitive argument required
    auto b1 = drv::sin(b_);
    r_ = drv::atan2(a1, b1); // at least one non-primitive req.
}

std::cout << a_drv << std::endl;
std::cout << b_drv << std::endl;
```

Permit mixed modes

- Operator overloading AD tools typically only accommodate the case where there can be only one permissible active type in a given context.

- Passive inputs may be made so by leaving their type as primitive, such as float.

- If, however, both a fully differentiated and a partially differentiated function are wanted, two versions of the code would be needed...

- ... unless we permit the co-existence of different active types, i.e. primal and tangent or primal and adjoint.
Expression node tagging

```cpp
Drv<Primal, float> a_{a, a_drv}; // passive but not primitive
Drv<Adjoint, float> b_{b, b_drv};
Drv<Adjoint, float&> r_{r_drv};
{
    auto a1 = drv::cos(a_); // non-primitive argument required
    auto b1 = drv::sin(b_);
    r_ = drv::atan2(a1, b1); // at least one non-primitive req.
}
```

Expression types:

```cpp
{
    Unary<Primal, Cos, Drv<Primal, float>> // a1
    Unary<Adjoint, Sin, Drv<Adjoint, float>> // b1
    Binary<Adjoint, ATan2, // r_ rhs
        Unary<Primal, Cos, Drv<Primal, float>>
        Unary<Adjoint, Sin, Drv<Adjoint, float>>}
```
Tag promotion

– If \( a1 \) were a primitive then \( \text{drv} :: \cos(a1) \) would cause a compilation failure; unary operators only accept an active type.

– The tag for the expression \( \text{drv} :: \text{atan2}(a1, b1) \) is deduced from its child nodes using promotion rules.
Compile time identity
Can I query, at compile time, if the left branch identical to the right branch?

*Apparently not, despite it being statically obvious.*

```cpp
template<typename A, typename B>
auto compute(A &&a, B &&b)
{
    float c0 = ...
    float c1 = ...

    auto ab = a * b;
    auto left = c0 + ab;
    auto right = c1 + ab;
    return left / right;
}
```
Same type, different names
Unique types

template<typename T, auto = [](){}> // lambda type is unique
struct Unique { T value; };

Unique<float> c0;
Unique<float> c1;

static_assert(!std::is_same_v<decltype(c0), decltype(c1)>);
Unique ‘addresses’

template<typename E0, typename E1>
auto constexpr isSame(E0 &&e0, E1 &&e1)
{
    return &e0 == &e1;
}

float c0;
float c1;

static_assert(isSame(c0, c0));
static_assert(!isSame(c0, c1));
Unique ‘addresses’: no good!

template<typename OP, typename E0, typename E1, bool IsSame>
struct Binary { ... };

template<typename E0, typename E1>
auto add(E0 &&e0, E1 &&e1)
  -> Binary<Add, E0, E1, IsSame(e0, e1)>
{ ... }

float c0;
float c1;
add(c0, c1); // error: non-type template argument
// is not a constant expression
C++  SG7 discussions

– I made a proposal to SG7, ‘varid operator’, December 2020, which drew some discussion.

– Andrew Sutton proposed a major revision of how reflection should be done around the same time. In P2237R0, ‘Metaprogramming’, p. 15, this paper describes a function, meta::location_of(expr), returning line and column values.

– Marco Foco, et al, ‘P2072R1: Differentiable programming for C++’ provides a summary of the state of the art and a presentation on desirable language support.

– C++20 added std::source_location. I’m not sure how useful it is for this problem...
Summary
Summary

– An eclectic approach to differentiation has been taken, which is limited to what the language features permit, but excludes taping.

– The approach generally requires code to be written in a ‘pure functional’ manner.

– Template expressions are employed at the fine grain level, whereas at the coarse grain less formal approaches are used.

– Finding ways of constructing the efficient adjoint code within C++ would greatly help, even if this means influencing language changes. The reliance on taping is, in my opinion, a dead end.
Block sequencing, partial derivatives and SG7
24\textsuperscript{th} EuroAD Workshop

Dominic Jones

Siemens, London
dominic.jones@cd-adapco.com

2–4 November 2021